WAIT, THERE'S MORE IN ERLANG

As if it wasn't enough to be on par with most languages already, Erlang's got yet another error handling structure. That structure is defined as the keyword `catch` and basically captures all types of exceptions on top of the good results. It's a bit of a weird one because it displays a different representation of exceptions:

```
1> catch throw(whoa).
whoa
2> catch exit(die).
{'EXIT',die}
3> catch 1/0.
{'EXIT',{badarith,[{erlang,'/',[1,0]},
{erl_eval,do_apply,5},
{erl_eval,expr,5},
{shell,exprs,6},
{shell,eval_exprs,6},
{shell,eval_loop,3}]}}
4> catch 2+2.
4
```

What we can see from this is that throws remain the same, but that exits and errors are both represented as`{'EXIT', Reason}`. That's due to errors being bolted to the language after exits (they kept a similar representation for backwards compatibility).

The way to read this stack trace is as follows:

```
5> catch doesnt:exist(a,4).
{'EXIT',{undef,[{doesnt,exist,[a,4]},
{erl_eval,do_apply,5},
{erl_eval,expr,5},
{shell,exprs,6},
{shell,eval_exprs,6},
{shell,eval_loop,3}]}}
```

- The type of error is `undef`, which means the function you called is not defined (see the list at the beginning of this chapter)
- The list right after the type of error is a stack trace
- The tuple on top of the stack trace represents the last function to be called (`{Module, Function, Arguments}`). That's your undefined function.
- The tuples after that are the functions called before the error. This time they're of the form `{Module, Function, Arity}`.
- That's all there is to it, really.

You can also manually get a stack trace by calling `erlang:get_stacktrace/0` in the process that crashed.

You'll often see `catch` written in the following manner (we're still in exceptions.erl):

```
catcher(X,Y) ->
case catch X/Y of
{'EXIT', {badarith,_}} -> "uh oh";
N -> N
end.
```

And as expected:

```
6> c(exceptions).
{ok,exceptions}
7> exceptions:catcher(3,3).
1.0
8> exceptions:catcher(6,3).
2.0
9> exceptions:catcher(6,0).
"uh oh"
```

This sounds compact and easy to catch exceptions, but there are a few problems with `catch`. The first of it is operator precedence:

```
10> X = catch 4+2.
* 1: syntax error before: 'catch'
10> X = (catch 4+2).
6
```

That's not exactly intuitive given that most expressions do not need to be wrapped in parentheses this way.

Another problem with `catch` is that you can't see the difference between what looks like the underlying representation of an exception and a real exception:

```
11> catch erlang:boat().
{'EXIT',{undef,[{erlang,boat,[]},
{erl_eval,do_apply,5},
{erl_eval,expr,5},
{shell,exprs,6},
{shell,eval_exprs,6},
{shell,eval_loop,3}]}}
12> catch exit({undef, [{erlang,boat,[]},
{erl_eval,do_apply,5}, {erl_eval,expr,5}, {shell,exprs,6},
{shell,eval_exprs,6}, {shell,eval_loop,3}]}).
{'EXIT',{undef,[{erlang,boat,[]},
{erl_eval,do_apply,5},
{erl_eval,expr,5},
{shell,exprs,6},
{shell,eval_exprs,6},
```

`{shell,eval_loop,3}]}}`

And you can't know the difference between an error and an actual exit. You could also have used `throw/1` to generate the above exception. In fact, a `throw/1` in a `catch` might also be problematic in another scenario:

```
one_or_two(1) -> return;
one_or_two(2) -> throw(return).
```

And now the killer problem:

```
13> c(exceptions).
{ok,exceptions}
14> catch exceptions:one_or_two(1).
return
15> catch exceptions:one_or_two(2).
return
```

Because we're behind a `catch`, we can never know if the function threw an exception or if it returned an actual value! This might not really happen a whole lot in practice, but it's still a wart big enough to have warranted the addition of the `try ... catch` construct in the R10B release.

Source : http://learnyousomeerlang.com/errors-and-exceptions